Short note

read_graph_from_file1

The code is pretty self-explanatory, but anyways. In addition to parsing the number of nodes `N` and skipping the first four lines in the text file, the program reads each pair of integers and checks that their values are legal. I.e, it checks that they are not equal (a node cannot be connected to itself), that they are non-negative and that they do not exceed the total number of nodes `N`. If their values are legal, the corresponding elements in `table2D` are set equal to 1.

Algorithm:

Initialize `NxN` matrix `table2D` with zeros

Declare `FromNodeId, ToNodeId`

for each line of integers in text file:
    Store each pair of integers in `FromNodeId, ToNodeId`
    if the values are legal:
        `table2D[FromNodeId][ToNodeId] = 1`
        `table2D[ToNodeId][FromNodeId] = 1`

----

read_graph_from_file2

We again read through each pair of integers and check that their values are legal. If so, the corresponding elemments of `row_ptr` are incremented by 1. As `row_ptr` is an array of `N+1` elements with the zeroth element always being a zero, the corresponding element of the node with index `NodeId` is `row_ptr[NodeId+1]`. The elements of `row_ptr` are then summed up cumulatively such that `row_ptr` essentially counts the number of 1's in each row of `table2D` in a cumulative way. Since the number of 1's in `table2D` is twice the number of edges in the connectivity graph, the last element of `row_ptr` is the length of `col_idx.`

`row_ptr` now contains information about how many edges each node has, so that we know the "range" of each node in `col_idx`. In the example given in the problemt text, node 0 has 3 edges so that the range of node 0 in `col_idx` is the first three elements, (the range of node 1 is the next three elements, and so on). Formally, the range of each node is given by the elements of `col_idx` between `col_idx[row_ptr[NodeId]]` and `col_idx[row_ptr[NodeId]`

+ NodeIdEdges] where NodeIdEdges are the number of edges of NodeId.

We again read through pairs of integers in the text file and check that their values are legal. If so, we store ToNodeId in the range of FromNodeId in col_idx, and vice versa.

Algorithm:

```
-------------------------------------------------------------------
Initialize N+1 array row_ptr with zeros

Declare FromNodeId, ToNodeId

for each line of integers in text file:
        Store each pair of integers in FromNodeId, ToNodeId
        if the values are legal:
                Increment row_ptr[FromNodeId+1] by 1
                Increment row_ptr[ToNodeId+1] by 1

for i=2:N-1
        row_ptr[i] = row_ptr[i] + row_ptr[i-1]

Declare row_ptr[N] array col_idx

for each line of integers in text file:
        Store each pair of integers in FromNodeId, ToNodeId
        if the values are legal:
                Store FromNodeId in the range of ToNodeId in col_idx
                Store ToNodeId in the range of FromNodeId in col_idx
-------------------------------------------------------------------
```

create_SNN_graph1

SNN_table is essentially given by the matrix product of table2D with itself, except that rows and columns corresponding to two unconnected nodes are ignored (since the similarity between unconnected nodes is 0 by default). The matrix product in index notation is given by

$$C_{ij} = \sum_k A_{ik} B_{kj}.$$

Since we have a matrix product of a matrix with itself, we can write

$$B_{ij} = \sum_k A_{ik} A_{kj}.$$

The matrix is symmetric, allowing us to substitute $A_{kj} \to A_{jk}$. This is more efficient since C/C++ uses column major order when storing arrays in memory (subsequent matrix elements in each row lie next to each other on the cache line). Furthermore, as `SNN_table` is itself symmetric and with zeros along the diagonal, we only have to compute the upper triangular elements. We can then set the lower triangular elements accordingly.

Algorithm:

```
Initialize NxN matrix SNN_table with zeros

for i=0:N-1
      for j=i+1:N-1
            if nodes i and j are connected
                  for k=0:N-1
                        SNN_table[i][j] = SNN_table[i][j] + table2D[i]
[k]*table2D[j][k]
                  SNN_table[j][i] = SNN_table[i][j]
```

---

create_SNN_graph2

As I couldn't figure out how to do symmetric matrix multiplication in CRS format, the implementation in create_SNN_graph2 is pretty brute force. It loops over all nodes, where it again loops over all nodes connected to each node, and simply counts how many nodes they are both connected to.

Algorithm:

Initialize row_ptr[N] array SNN_val with zeros

Declare node1, node2, node3, node4

```
for node1=0:N-1
        for each node node2 connected to node1
                for each node node3 connected to node2
                        for each node node4 connected to node1
                                if node3=node4
                                        increment the element corresponding to node2 in the range of node1 in SNN_val
```

----

check_node

Here I took inspiration from the breadth-first search algorithm (see https://en.wikipedia.org/wiki/Breadth-first_search) which is used for searching graph data structures from a given "root". I used a modification which simply prints out the nodes that meet the threshold tau.

Algorithm:

Declare vector queue

Initialize N array discovered with zeros

Declare node, discovered_node, tau

```
Add node_id to queue
while queue is not empty
        remove first element of queue and store it in node
        for all nodes discovered_node connected to node
                if discovered[discovered_node]=0 and similarity between discovered_node and node is at least tau
                        print discovered_node
                        discovered[discovered_node] = 1
                        add discovered_node to queue
```
